Generating Query Templates for a Personalized Service Level Agreement in the Cloud

Alekzander Malcom, Jennifer Ortiz

ABSTRACT

Public Clouds today provide a variety of services for data analysis such as Google BigQuery and Azure. Each service comes with a pricing model and service level agreement (SLA). Today's pricing models and SLAs are described at the level of compute resources (instance-hours or gigabytes processed). They are also different from one service to the next. Both conditions make it difficult for users to select a service, pick a configuration, and predict the actual analysis cost. To address this challenge, we propose a new abstraction, called a *Personalized Service Level Agreement*, where users are presented with what they can do with their data in terms of query capabilities, guaranteed query performance and fixed hourly prices. In this paper, we primarily focus on exploring the search space of potential query capabilities and time thresholds based on a Cloud service.

1. INTRODUCTION

Many data management systems today are available as Cloud services. For example, Amazon Web Services (AWS) [1] include the Relational Database Service (RDS) and Elastic MapReduce (EMR); Google offers BigQuery [4]; and SQL Server is available on Windows Azure [2]. Each service comes with a pricing model that indicates the price to pay based on the level of service.

An important challenge with today's pricing models is that they force users to translate their data management needs into resource needs (How many instances should I use? How many gigabytes will my queries process?). There is thus a disconnect between the resource-centric approach expressed by Cloud providers and what the users actually wish to acquire [7]. The knowledge required to understand the resources needed for data management workloads is a challenge – particularly when a user does not always have a clear understanding of their data or even know what they are looking for [8]. Furthermore, pricing models can be wildly different across providers [9]. For example, Azure charges for the size of compute instances acquired, while BigQuery charges per GB processed by a query. This heterogeneity complicates the decision behind selecting a service.

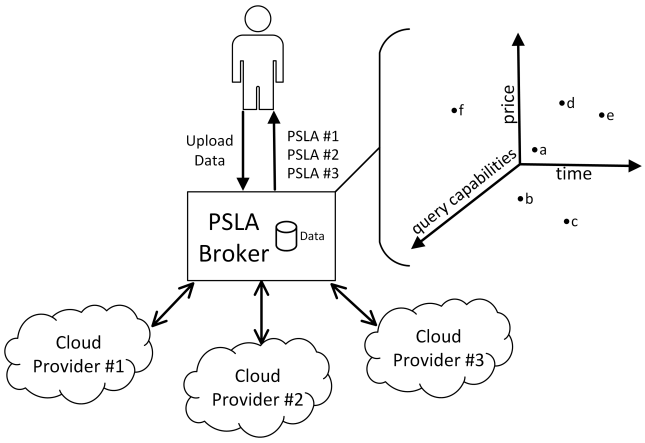

Figure 1: The PSLA system acts as a broker between different Cloud-provided data management services and the user. For each service, it generates alternative agreements with different trade-offs between cost, performance (query runtimes), and query capabilities. In this paper, we focus on the query capabilities and time axes.

As a second challenge, although Cloud providers offer availability through service level agreements (SLAs), they do not provide any type of performance guarantee. Studies suggest that there exist variances in the performance of jobs executed in Cloud services [10].

In this paper, we focus on providing users a different way to interact with their Cloud Service provider. Instead of asking the user to specify the resources that she wants or asking the user for the queries she needs to execute, our proposed system looks at the user's data and shows the user what she can do with the data for a set price. The focus of our proposal is to ensure both high-performance and simplicity, two core features required by Cloud users [14]. As shown in Figure 1, the key idea is for a user to simply upload her data to a Cloud broker system and be presented with service-level alternatives that correspond to different points in the space by trading off *query runtime performance*, *monetary cost*, and the *query capabilities* of the service. We call these agreements, personalized SLAs or PSLAs, because their detailed specification depends on the data uploaded by the user. In this paper, we identify different alternative methods to automate the generation of such a PSLA. We begin by exploring the query capabilities and time search space. For now, we predetermine the price by hand-picking a service level from each Cloud provider.

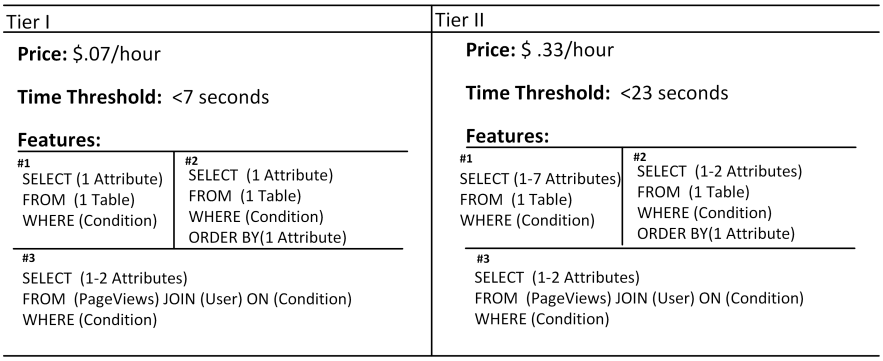

Figure 2: Example of a potential PSLA for BigQuery. Here the user decides between price, query capabilities and time thresholds

The query capability features are demonstrated in the form of templates for the types of queries that the user can execute. While prior work studied techniques that enable users to specify a desired price-performance curve when submitting a query [15], the approach simply rejects queries whose performance cannot be satisfied. Instead, we focus on informing users about what they can and cannot do with their data within specified performance bounds. An example of such a PSLA is shown in Figure 2.

For this paper, we focus on the query performance and query capability dimensions of the problem. As an initial step, we focus on PSLAs for structured and relational data only.

2. MOTIVATION

We begin with a motivating scenario to demonstrate the challenges with today's Cloud pricing models and SLAs.

2.1 Motivating Example

Sarah is a data scientist with access to a log which captures all of the Web pages viewed by a set of users over some period of time. This scenario corresponds to the PigMix benchmark [13]. Sarah has access to 100GB of data but decides to explore a 10GB subset. The schema for the data is as follows:

Users (name, phone, address, city, state, zip)

PageViews(user, action, timespent, query_term, ip_addr, timestamp, estimated_revenue)

Sarah's first choice is between using BigQuery or SQL Server on Azure. In both cases, she will be able to study her data by issuing SQL declarative queries. The challenge, however, is that the trade-off between price, performance, and capabilities for each service is not directly obvious from the online pricing models. For example, from the pricing scheme, it is unclear how many GB she will need to process in order to effectively explore her data. Thus, an accurate estimate for the end cost is nearly impossible. She will also need to keep in mind that there are certain limitations to BigQuery's SQL-like language. For example, operators such as DISTINCT are disallowed. Although BigQuery provides Sarah the ability to interactively query her data, there is no guarantee of how long it will take a query to return a response.

On the other hand, SQL Server on Azure would charge Sarah based on the number of virtual machines that are initialized and the amount of compute time (per hour). If she selects a small machine with SQL Server, the cost would be $0.08 per hour while a large machine would cost up to $0.32 per hour. Would selecting a different instance size help Sarah get faster results? Additionally, if Sarah does not know the type of workload she wishes to run, which service should she pick?

This example illustrates that, today, it is difficult for users to select a Cloud provider and level of service that corresponds to a desired trade-off between cost, performance, and query capabilities. Users today need to proceed by trial and error when selecting service levels. Even for experts, finding cost-effective methods to process large datasets continues to be a nontrivial task [3].

2.2 PSLAs

To address the above challenge, we propose to re-think the interface between users and Cloud services. Instead of forcing users to translate their price, performance, and capability goals into resource requests, we propose to automatically perform this translation for them. In particular, we want users to simply upload their data. The system should *automatically analyze that data* (eg. compute statistics on the data) and describe to users what they can and cannot do with their data based on price and performance factors. We call these descriptions Personalized Service Level Agreements (PSLAs). We first define a PSLA more precisely. We then discuss the challenges associated with generating such PSLAs automatically.

A PSLA is composed of a set of tiers $R_1, R_2....R_k$. Each tier corresponds to a unique level of service. That is, each tier offers a specific trade-off between query capabilities, price, and performance. No tier strictly subsumes another in the same PSLA. Each tier R_i, $1 \leq i \leq k$, has an hourly price P_i, a time threshold T_i, and a set of query templates $\{M_{i1}, M_{i2}, ..., M_{iv}\}$. Query templates define the capabilities available in the tier. The time threshold T_i guarantees that all queries which follow the templates will return within the specified time. The user selects one tier from the set. Specifically, a PSLA is a set of the form:

$$\begin{aligned} PSLA = \{&R_1 = (P_1, T_1, \{M_{11}, M_{12}, ..., M_{1v}\}),\\ &R_2 = (P_2, T_2, \{M_{21}, M_{22}, ..., M_{2v}\}),\\ &...,\\ &R_k = (P_k, T_k, \{M_{k1}, M_{k2}, ..., M_{kv}\})\} \end{aligned}$$

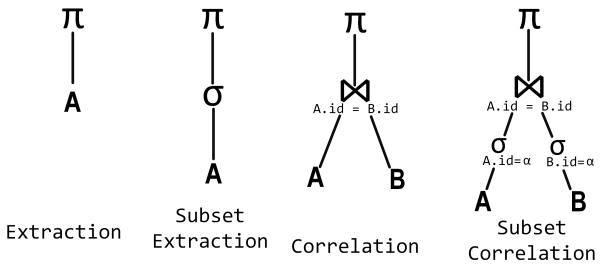

Figure 3: Four query categories: Extraction, Subset Extraction, Correlation, Subset Correlation

To generate the query templates for a given dataset, we consider a few relational operators including selection, projection and join. We only consider simple queries that use a small number of these operators at the same time. We assume there are no indexes, and consider only explicitly specified primary and foreign key constraints.

2.3 Research Challenges

The PSLA system should *automatically analyze the data* (eg. compute statistics on the data) and describe to users what they can and cannot do with their data based on performance factors. The challenge here is finding the type of statistics needed from the data. This process needs to be quick since we do not users to wait too long before the system displays a PSLA.

Additionally, given an input database comprising a set of relations, the number of queries that can be posed over the data is unbounded. The PSLA system, however, must show users a bounded set of query templates through each service tier. The key question is how to select *good* query templates. A small number of templates would facilitate tier comparisons for the user. Furthermore, based on prior experience working alongside domain scientists, we find that providing templates even for simple queries covers a significant amount of data processing needs.

3. APPROACH

In this section, we focus on explaining how we generate tiers based on a given dataset. Once we generate the queries needed, we cluster the queries based on estimated costs. Lastly, we discuss how to translate the clustering results into a set of PSLA tiers for the user.

In order to determine a bounded set of possible queries that can be generated from a dataset, we begin by assuming a star schema dataset. A star schema dataset contains a fact table F and a set of k dimension tables D, where D=$\{d_1, d_2, ...d_k\}$. In terms of join relationships between the F and some d_i, there are no cycles, but there exists at most one self-join for each d_i table.

3.1 Query Generation and Data Statistics

We consider four query categories to generate. These queries consist of selection, projection and join operators. As seen in Figure 3, we identify these four categories as Extraction, Subset Extraction, Correlation, and Subset Correlation. Queries that are of the form Extraction consist of a projection. Subset Extraction queries represent those composed of one table along with a selection and projection. For Correlation queries, these consist of a projection and join operator that joins on matching attributes (i.e. either on a primary key/foreign key between F and some d_i or on the same key if it is a self-join). The final is the Subset Correlated query.

In order to understand the reasoning behind the Subset Correlated query category, we must first discuss the type of selections we wish to generate. For Subset Extraction queries, we consider queries that contain predicates that will result in a percentage of tuples based on orders of magnitude, specifically 100%, 10%, 1%, and .01%. In order to generate an adequate predicate to fit that criteria, we computed a histogram that displays each attribute in sorted order. This histogram helps us rank the attributes and thus, we can select a predicate that will fit the desired order of magnitude.

In the case for the Subset Correlation category, we want to generate selections on joins. If we are to select predicates that also result to a value close to the desired order of magnitude, we must provide a way to determine the selectivity without actually computing the join (for efficiency purposes).

According to the work by Chaudhuri, Motwani and Narasayya in [5], we must be cautious when sampling from joins. They describe how, for example, the join of random samples from a relation R_1 and relation R_2 does not give a random sample of the join $R_1 \bowtie R_2$. In other words, sampling does not commute with joins. The naive strategy for sampling from a join would be to compute the full join and then sample therein [5]. Instead, we wish to avoid computation of joins in order to save time. Therefore, we make use of the histograms we generate and filter both tables by the attribute we join on. For example, in Figure 3, we filter relation A and B under Subset Correlation based on the join attribute. In this case, the predicate would consist of some α based on the selectivity we wish to acquire. Assuming a star schema and the implications explained under the approach section, the query generation algorithm is described below. This method was implemented through a python script by having the generated histogram as an input.

Data: histogram, F and set D
Result: A set of queries Q
Emulate schema and provide text aliases for repeated table instances;
k = number of tables = $|D|$;
for *each* n *in* $\{0, 1, \ldots, 2k+1\}$ **do**
  for *each* J *a weak_composition*(n, k) **do**
    `/* J is a mask determining which tables to join */`
    if $J_0 = 0$ **then** `/* F does not appear */`
      Disregard J if more than one index i has $J_i > 0$;
    end
    if *any* $J_i > 2$ **then**
      Disregard J
    end
    T = join of J_0 copies of F, J_1 copies of d_1, etc.;
    Sort attributes of T by average size of data in bytes;
    G = number of attributes of T;
    for *each* $g \in \{1, 2, \ldots, G\}$ **do**
      yield a query selecting largest g attributes from T;
      yield a query selecting smallest g attributes from T;
    end
  end
end

Algorithm 1: Query Generation Algorithm

We should note that the form of the weak composition problem needed here, i.e. to find integers $x_i \geq 0$ such that $\sum_{i=1}^{k} x_i = n$ has a very simple recursive solution.

Data: integer n to be partitioned, integer k parts
Result: list of lists of non-negative integers, each of which sums to n and has length k
if $k = 1$ **then**
  yield List(n)
else
  for *each* i *in* $\{0, 1, \ldots, n\}$ **do**
    for *each* X *a weak_composition of* $n - i$ *into* $k - 1$ *parts* **do**
      yield X+List(i)
    end
  end
end

Algorithm 2: Weak Composition

3.2 Clustering Estimated Query Costs

Before generating tiers for the PSLA, we must gain a sense of the performance for each query. Once the queries are generated, we run all the queries on different levels of service through Azure. We acquired one small virtual instance (1 virtual core, 1.75 GB Ram) and a large virtual instance (4 virtual cores, 14 GB Ram) where both instances include SQL Server 2008.

Assuming we wish to utilize the estimated costs of the queries to determine the PSLA tiers, we need to first determine whether the estimated costs from the SQL Server optimizer really reflect the query runtimes. For each machine, we cluster for the query runtimes and for the estimated query costs. We use WEKA [6] to cluster results based on k-means algorithm.

In order to determine whether there exists a similarity between the real query times and the estimated costs, we utilize a criterion called Variation of Information (VI) from Meila's work in [11]. This criterion measures the amount of information lost or gained in changing from clustering $\mathcal{C}$ to clustering $\mathcal{C}'$. The paper refers to a clustering as a partition of a set of points into sets $C_1, C_2, \ldots, C_k$. In our case, we refer the query runtimes as $\mathcal{C}$ and the estimated query times as $\mathcal{C}'$. The VI equation takes into account the entropy associated with each clustering referred to as $H(\mathcal{C})$ and $H(\mathcal{C}')$ as well as the reduction in uncertainty based on given knowledge as expressed through $I(\mathcal{C}, \mathcal{C}')$. We display the equation below. For more details about this criterion, see [11].

$$VI(\mathcal{C}, \mathcal{C}') = H(\mathcal{C}) + H(\mathcal{C}') - 2I(\mathcal{C}, \mathcal{C}')$$

VI will always result in a non-negative value (further, VI is a metric on the space of all clusterings). One property of the VI equation states that if both clusterings are the same, then the result will be 0. The upper bound (or maximum) between two clusterings is $VI(\mathcal{C}, \mathcal{C}') \leq 2 \log K$, where K represents the maximum number of clusters between $\mathcal{C}$ and $\mathcal{C}'$, and provided that K^2 is less than the number of data points.

3.3 Cluster to Tier Translation

In this section, we describe a method to translate the clusters from estimated query compute costs into tiers. Each tier will be composed of a set of query templates. First, we define a query template more precisely. We use this definition to distinguish between query templates.

Definition Let a set of query templates $M_1, M_2, ...M_k$ represent the queries that are shown to the user. Each query template, M_i, takes the form of one of the query categories described from Figure 3. Specifically a query template consists of a category, a set of tables, and a selectivity level (either 100%, 10%, 1%, or .01%).

3.3.1 Tier Generation Algorithm

In this algorithm, we focus on generating a set of tiers T from a given clustering $\mathcal{C}$. Each cluster C_i in the clustering consists of a set of points $P_1, P_2, \ldots, P_k$. For each cluster we iterate through, we will generate a new tier and add the query template to the current tier. However, if the query template exists in the current tier or a previous tier, it is not added to the current tier. Below, we describe the algorithm. We run this algorithm on each estimated query compute cost of each service level (small and large) and provide all the resulting tiers to the user. In this algorithm, the abstraction of a point P_i is defined as a function that returns the query template that the point represents. We iterate through all the points in each cluster and add the abstraction of P_i to a tier. As we iterate, if P_i is contained in a query in the current tier, we disregard the point and do not add it to the tier. Currently, this process is done manually.

Data: A Clustering $\mathcal{C}$
Result: A set of Tiers
sort clusters descending by average estimated cost ;
initialize $T = \emptyset$;
for *each cluster* C_i *in clustering* $\mathcal{C}$ **do**
  initialize tier R_i;
  for *each* P_i *in* C_i **do**
    Q = $abstractform(P_i)$;
    if $Q \notin R_i \wedge Q \notin T$ **then**
      if $\exists Q' \in R_i$ *where* $Q \sqsubseteq Q'$ **then**
        Discard Q;
      else
        R_i = $R_i \cup \{Q\}$;
      end
    end
  end
  if $R_i \neq \emptyset$ **then**
    yield R_i;
    T = $R_i \cup T$;
  end
end

Algorithm 3: Tier Generation Algorithm

4. RESULTS

In this section we describe the results for Azure small instance and Azure large instance based on the approach described in the previous section.

By using 10GB Pigmix dataset, the query generation algorithm generates a total of 460 queries. We cannot expect to both run all the queries in an adequate amount of time and provide the user with a PSLA thereafter. In fact, running all the queries generated on the small Azure instance took 2312180ms while it took 2504045ms on the large instance. Instead, we look at the estimated query cost from the optimizer. The SQL Server optimizer provides a type of metric that can measure a cost based on CPU and I/O cost that a query may utilize. Estimating the compute cost of all the queries on the small instance took 7168ms while on the large instance it took 5668ms.

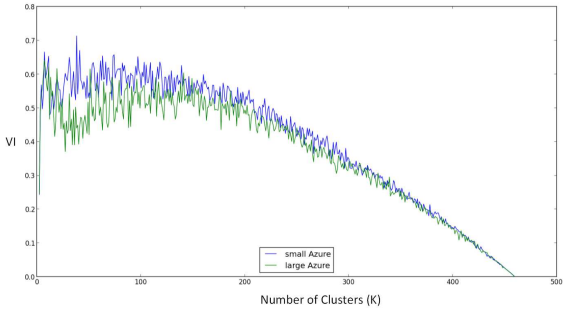

Figure 4: VI for Increasing K Values

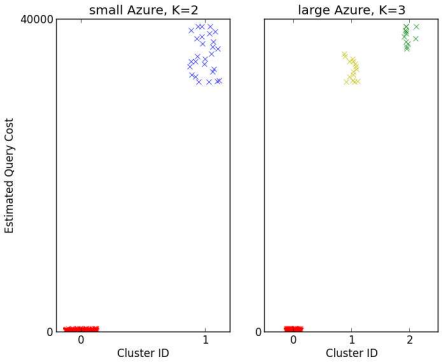

Figure 5: Clusters

4.1 VI Dependence on K

In this section we discuss the best k value we wish to cluster by for both small Azure and large Azure instances. In theory, we wish for the value of VI to be as low as possible since a smaller VI represents more similar clusters. However, the lowest VI value (i.e. zero) occurs when $k = n$, where n is the number of points, and according to our experiment, this can only reliably happen then. It would be unrealistic to expect a user to interpret n tiers, one for each of these clusters. Thus, we consider only a small number of tiers, by restricting $k \leq 10$; we arbitrarily assume that the user is interested in at most 10 tiers. In Figure 4, we can see the value of VI fluctuate for small values of k, only to slowly decrease to zero thereafter. In the tables below, we inspect the VI values of Azure Small instance and Azure Large instance more closely; we have truncated the list of values as the VI increases greatly after $k = 4$.

Azure Small	
k=2	VI =.099
k=3	VI =.29
k=4	VI =.39

Azure Large	
k=2	VI =.245
k=3	VI =.25
k=4	VI =.43

From these charts, we select the lowest K value. For small Azure, we will focus on the clustering given by k = 2. While for large Azure, we will look at the clustering given by k = 3. Refer to Figure 5.

4.2 Generated Tiers

For small Azure and large Azure, we manually generated the tiers based on the algorithm described in the previous section. Below, we display the tiers generated for each instance. The times are given by running one representative query (the upper bound) of the belonging cluster. At times when the representative query took too long to return all the answer tuples, we only selected the TOP (1000000) results. We reflected this change in the templates. The resulting templates are shown in Figures 6 and 7.

4.3 Total Generation and Cluster Time

In this section, we describe all the steps and time taken in order to generate query templates. We wish to keep track of the time since our goal is to provide quick PSLA tier results to the user. Below we list each step and the resulting time.

- Creating Histogram: .013s (Small) and .014s (Large)
- Reading Histogram: 225s
- Generating Queries: .1225s
- Reading Estimated Costs for All Queries: 7.168s (Small) and 5.668s (Large)
- Clustering Estimated Costs and Exploring VI Values: 2.21s
- Finding Representative Queries: .47s (Small) and .61s (Large)
- Running Representative Queries: 80s (Small) and 56s (Large)

The total time to generate these steps for the small Azure instance takes ~314.98 seconds. While it takes ~289.62 seconds for the large Azure instance. As we can see, the current main bottleneck is reading the histogram to generate the queries. In the future, this will be optimized. In the next section, we evaluate the generated tiers by running it through some random sample queries.

5. DISCUSSION

In this paper, we have demonstrated a method to automatically generate a set of queries from a given data set. Upon clustering the queries, we can develop a set of tiers that display the types of queries that can be run under a certain specified time threshold.

5.1 Tier Evaluation

We randomly created 20 queries that follow the form of the templates shown in Figures 6 and 7. We ran these queries for the large and small Azure instances. All the queries were able to run under the required threshold.

However, there is much work that needs to be done to improve upon these PSLA tiers. For instance, the tiers shown for the large Azure instance in Figure 7, have really close and similar time constraints. Technically, we could potentially merge all these three tiers into one. An alternative approach would be to generate more expensive queries in order to see a wider selection of unique tiers.

What is interesting about these templates is that it provides an easy way to compare between two types of machines. For instance, if a user were to come upon these tiers, they could quickly notice how they should pick Tier II over Tier V. Both of these tiers provide the same query capabilities and time threshold, but one is much

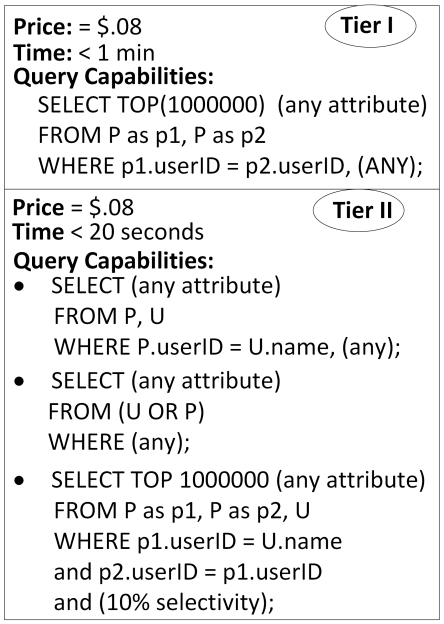

Figure 6: Tiers for Azure Small Instance

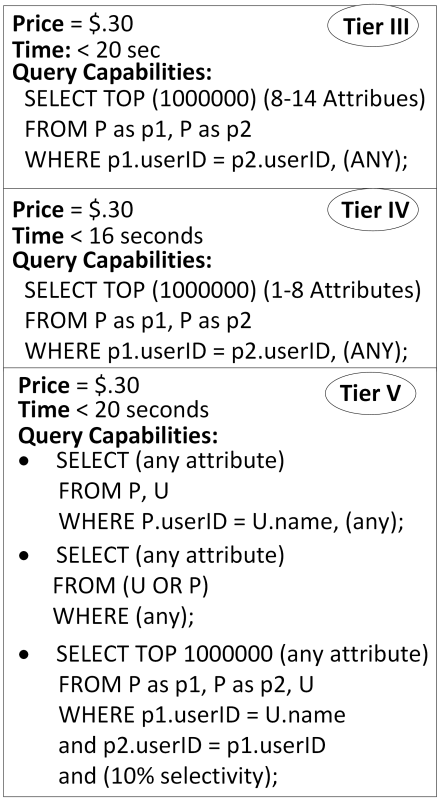

Figure 7: Tiers for Azure Large Instance

cheaper than the other. On the other hand, if the user wants to run queries over a self-join on the PageViews table (P), the best option would be to select Tier III. Although this option is slightly more costly, we will acquire our results faster.

Additionally, running these queries on a local instance of SQL Server is at times twice as fast as running them through Azure. Although this may seem obvious (i.e. we are not running queries through a Cloud service), it is still interesting to know about the query latency of a service ahead of time before actually paying for the service.

5.2 The Provider and the User

Overall, we show that one can easily derive meaningful service tiers that can help a user select a desired trade-off between price, performance, and capabilities. Most importantly, these service tiers enable high predictability: once a user pays a fixed price, she can run a pre-defined set of queries. This predictability benefits the user in that they know the performance capabilities ahead of time. For typical non-expert users, it is difficult to distinguish the performance capabilities between a small or large virtual machine. Additionally, the user will know how much the service will cost upfront. This results in more satisfied users who will not run into any cost surprises.

As for the Cloud provider, this service could provide more responsibility and yet, more flexibility. The Cloud provider must meet the PSLA given to the user which is a challenge. However, if the PSLA cannot be met, this could allow the provider to obtain more resources in order to meet the agreement. From an alternative perspective, if the provider knows ahead of time the type of queries the user wishes to run based on the PSLA selected, the provider can give more or less resources to the user while still meeting the agreement and not letting the user actually know the resources that were used.

6. RELATED WORK

With the rise of cloud computing, there has been an interest among large enterprises to adopt this type service to fulfill their data analytic needs. There are recent projects that have hinted towards the idea of re-evaluating what an SLA should consist of and how it should be presented to the user. For example, ActiveSLA [15] provides an admission control framework based on SLAs that try to maximize profit given the SLA rules and rejects queries that cannot meet SLA objectives. In contrast, our PSLAs tell users what they can and cannot do within different price-performance choices. Hence we never reject queries. Similarly, the DBSeer [12] work-inprogress project reaffirms how today's Cloud services do not properly address how to understand prices and resource utilization. They propose to create a model that can predict resource utilization based on a workload. This might work well for a OLTP type of workload, but in our project, we claim that the user does not always know what type of queries they want to run upfront, hence we automatically suggest workloads based on the data. Other work, as seen in the Bazaar framework [7] argue for a 'job-centric' cloud where tenants describe high-level goals regarding their jobs and the framework then can predict the resources needed to achieve the goal. Although this help users determine the cost and resource needs of their jobs, the users are required to describe their job by submitting a map-reduce job, a completion time and a price. This

would still require some type of expertise; instead we focus on users who only know basic SQL (such as domain scientists) and wish to use the Cloud to analyze the data.

7. CONCLUSION AND OUTLOOK

We present a the research challenges behind generating a selection of Cloud service tiers to a user. Our core idea is to examine a given dataset that a user wants to process and automatically generate *a personalized service level agreement(PSLA)*, which shows templates for the types of queries the user can execute on her data for a given price and within a given runtime threshold. PSLAs thus enable users to focus on their main concerns of query performance, cost, and capabilities, freeing them from the current resource-centric approaches. The next step in this project is to explore other methods of clustering to determine whether we can obtain a higher selection of interesting tiers. Additionally, we will focus on adapting our algorithms to fit a wider selection of more complex data models.

8. REFERENCES

[1] Amazon AWS. `http://aws.amazon.com/`.

[2] Azure. `http://www.windowsazure.com/en-us/`.

[3] S. Babu. Towards automatic optimization of MapReduce programs. In *Proc. of the First SoCC Conf.*, pages 137–142, 2010.

[4] Google BigQuery. `https://developers.google.com/bigquery/`.

[5] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, Proc. of the SIGMOD Conf., pages 263–274, New York, NY, USA, 1999. ACM.

[6] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[7] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the tenant-provider gap in cloud services. In *Proc. of the 3rd ACM Symp. on Cloud Computing*, pages 10:1–10:14, 2012.

[8] M. L. Kersten, S. Idreos, S. Manegold, and E. Liarou. The researcher's guide to the data deludge: Querying a scientific database in a just a few seconds. In *Proc. of the 37th VLDB Conf.*, volume 4, 2011.

[9] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: comparing public cloud providers. In *Proc. of the ACM SIGCOMM Conf.*, pages 1–14, 2010.

[10] W. Lu, J. Jackson, J. Ekanayake, R. S. Barga, and N. Araujo. Performing large science experiments on Azure: Pitfalls and solutions. In *IEEE CloudCom*, CloudCom'10, pages 209–217, 2010.

[11] M. Meila. Comparing clustering. 2002.

[12] B. Mozafari, C. Curino, and S. Madden. Dbseerl resource and performance prediction for building a next generation database cloud. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research CIDR 2013*, Proc. of the Sixth CIDR Conf., 2013.

[13] PigMix. Pigmix benchmark. https://cwiki.apache.org/PIG/pigmix.html, 2012.

[14] E. Wu, S. Madden, Y. Zhang, E. Jones, and C. Curino. Relational cloud: The case for a database service. Technical Report MIT-CSAIL-TR-2010-014, CSAIL MIT, 2010.

[15] P. Xiong, Y. Chi, S. Zhu, J. Tatemura, C. Pu, and H. HacigümüŞ. ActiveSLA: a profit-oriented admission control framework for database-as-a-service providers. In *Proc. of the Second SoCC Conf.*, pages 15:1–15:14, 2011.